ECE521 Inference Algorithms and Machine Learning
Report of Assignment 3

Group Members:
Xi Chen (Student ID: 1002266013)
Yang Wu (Student ID: 1002495132)

Task 1. K-means

1.1 Euclidean distance function

1.

$$D(\boldsymbol{X},\boldsymbol{Y}) = \begin{bmatrix} D(X_1,Y_1) & D(X_1,Y_2) & \cdots & D(X_1,Y_K) \\ D(X_2,Y_1) & D(X_2,Y_2) & \cdots & D(X_2,Y_K) \\ \vdots & \vdots & \ddots & \vdots \\ D(X_B,Y_1) & D(X_B,Y_2) & \cdots & D(X_B,Y_K) \end{bmatrix}_{B\times K}$$

$$= \begin{bmatrix} (X_1-Y_1)(X_1-Y_1)^T & (X_1-Y_2)(X_1-Y_2)^T & \cdots & (X_1-Y_K)(X_1-Y_K)^T \\ (X_2-Y_1)(X_2-Y_1)^T & (X_2-Y_2)(X_2-Y_2)^T & \cdots & (X_2-Y_K)(X_2-Y_K)^T \\ \vdots & \vdots & \ddots & \vdots \\ (X_B-Y_1)(X_B-Y_1)^T & (X_B-Y_2)(X_B-Y_2)^T & \cdots & (X_B-Y_K)(X_B-Y_K)^T \end{bmatrix}_{B\times K}$$

$$= \begin{bmatrix} X_1{X_1}^T & X_1{X_1}^T & \cdots & X_1{X_1}^T \\ X_2{X_2}^T & X_2{X_2}^T & \cdots & X_2{X_2}^T \\ \vdots & \vdots & \ddots & \vdots \\ X_B{X_B}^T & X_B{X_B}^T & \cdots & X_B{X_B}^T \end{bmatrix}_{B\times K} + \begin{bmatrix} Y_1{Y_1}^T & Y_2{Y_2}^T & \cdots & Y_K{Y_K}^T \\ Y_1{Y_1}^T & Y_2{Y_2}^T & \cdots & Y_K{Y_K}^T \\ \vdots & \vdots & \ddots & \vdots \\ Y_1{Y_1}^T & Y_2{Y_2}^T & \cdots & Y_K{Y_K}^T \end{bmatrix}_{B\times K} - 2\begin{bmatrix} X_1{Y_1}^T & X_1{Y_2}^T & \cdots & X_1{Y_K}^T \\ X_2{Y_1}^T & X_2{Y_2}^T & \cdots & X_2{Y_K}^T \\ \vdots & \vdots & \ddots & \vdots \\ X_B{Y_1}^T & X_B{Y_2}^T & \cdots & X_B{Y_K}^T \end{bmatrix}_{B\times K}$$

Where

- The first term can be derived from:

$$XX^T = \begin{bmatrix} X_1{X_1}^T & X_1{X_2}^T & \cdots & X_1{X_B}^T \\ X_2{X_1}^T & X_2{X_2}^T & \cdots & X_2{X_B}^T \\ \vdots & \vdots & \ddots & \vdots \\ X_B{X_1}^T & X_B{X_2}^T & \cdots & X_B{X_B}^T \end{bmatrix}_{B\times K} ;\ A = diag(XX^T) = \begin{bmatrix} X_1{X_1}^T & 0 & \cdots & 0 \\ 0 & X_2{X_2}^T & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & X_B{X_B}^T \end{bmatrix}_{B\times K}$$

Hence, $A \cdot \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}_{B\times K} = \begin{bmatrix} X_1{X_1}^T & X_1{X_1}^T & \cdots & X_1{X_1}^T \\ X_2{X_2}^T & X_2{X_2}^T & \cdots & X_2{X_2}^T \\ \vdots & \vdots & \ddots & \vdots \\ X_B{X_B}^T & X_B{X_B}^T & \cdots & X_B{X_B}^T \end{bmatrix}_{B\times K}$

Therefore, $D(\boldsymbol{X},\boldsymbol{Y})_1 = diag(XX^T)\cdot[1]_{B\times K} = diag(XX^T)\cdot I_{B\times K}$

- The second term can be derived in the similar process:

$$D(\boldsymbol{X},\boldsymbol{Y})_2 = (diag(YY^T)\cdot[1]_{K\times B})^T = (diag(YY^T)\cdot I_{K\times B})^T = {I_{K\times B}}^T \cdot diag(YY^T)^T$$

- The third term is simply as:

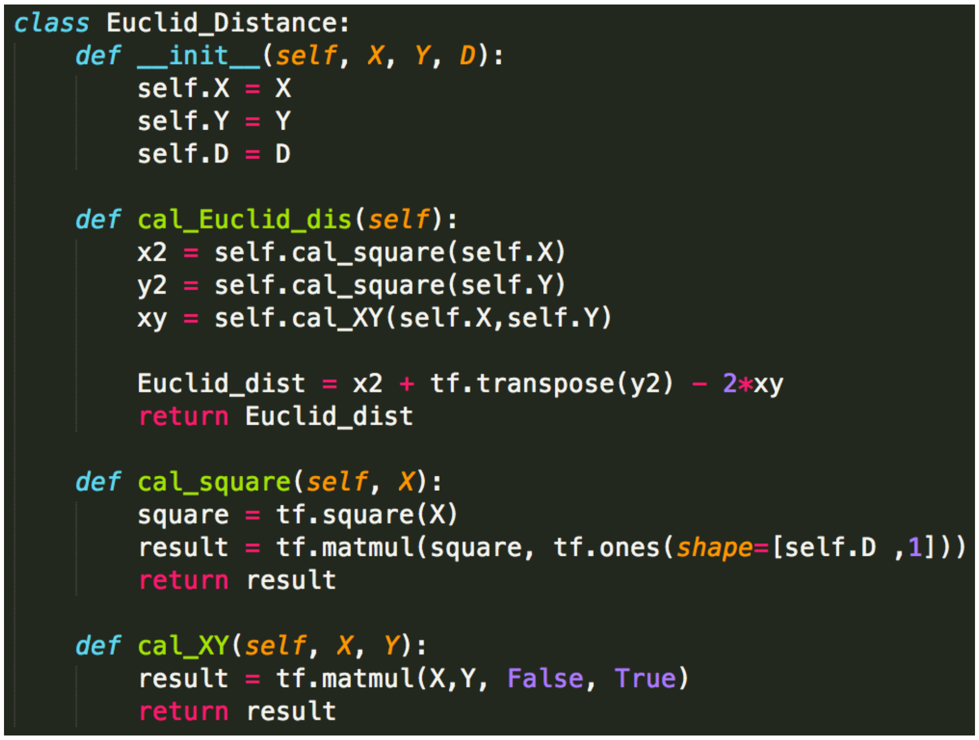
$$XY^T = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_{\boldsymbol{B}} \end{bmatrix}_{B\times D} \cdot [{Y_1}^T \quad {Y_2}^T \quad \cdots \quad {Y_K}^T]_{D\times K} = \begin{bmatrix} X_1{Y_1}^T & X_1{Y_2}^T & \cdots & X_1{Y_K}^T \\ X_2{Y_1}^T & X_2{Y_2}^T & \cdots & X_2{Y_K}^T \\ \vdots & \vdots & \ddots & \vdots \\ X_B{Y_1}^T & X_B{Y_2}^T & \cdots & X_B{Y_K}^T \end{bmatrix}_{B\times K}$$

$D(\boldsymbol{X},\boldsymbol{Y})_3 = 2 * XY^T$

Therefore, $D(\boldsymbol{X},\boldsymbol{Y}) = D(\boldsymbol{X},\boldsymbol{Y})_1 + D(\boldsymbol{X},\boldsymbol{Y})_2 - D(\boldsymbol{X},\boldsymbol{Y})_3$

$$= diag(XX^T) \cdot I_{B\times K} + {I_{K\times B}}^T \cdot diag(YY^T)^T - 2 * XY^T$$

2. The vectorized function has been written in the file "**Euclid_Distance.py**", which calculates Euclidean distance between possible pairs of vector points. The first two terms we derived above can be implemented by using Tensorflow broadcasting. The code is also listed as below:



```
class Euclid_Distance:
    def __init__(self, X, Y, D):
        self.X = X
        self.Y = Y
        self.D = D

    def cal_Euclid_dis(self):
        x2 = self.cal_square(self.X)
        y2 = self.cal_square(self.Y)
        xy = self.cal_XY(self.X,self.Y)

        Euclid_dist = x2 + tf.transpose(y2) - 2*xy
        return Euclid_dist

    def cal_square(self, X):
        square = tf.square(X)
        result = tf.matmul(square, tf.ones(shape=[self.D ,1]))
        return result

    def cal_XY(self, X, Y):
        result = tf.matmul(X,Y, False, True)
        return result
```

Fig1. Implements the pair-wise squared Euclidean distance function for two input matrices

1.2 Learning K-means

1. The loss function L(μ) is non-convex. If the function was convex, it would always converge to the global minimum; hence, the final value of L(μ) would be fixed under any circumstances. In our experiment, however, the loss function has different local minima and the final result depends on how the K cluster centers are initialized. The final result is only guaranteed to be one of these local minima, but it can be difficult to achieve the global minimum. Therefore, the loss function L(μ) is a non-convex function.

2. The functions are implemented in "**k_mean.py**". We set the learning rate as 0.001 and get the cluster centers are:

μ^1 [1.24792147, 0.25577992]
μ^2 [0.15492232, -1.51739335]
μ^3 [-1.05739701, -3.23398256]

Fig2. shows the loss vs. number updates.

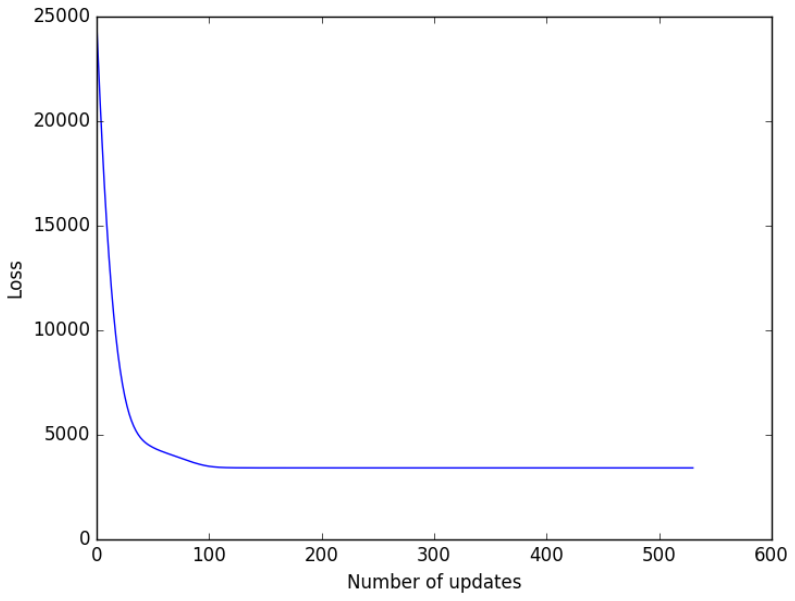
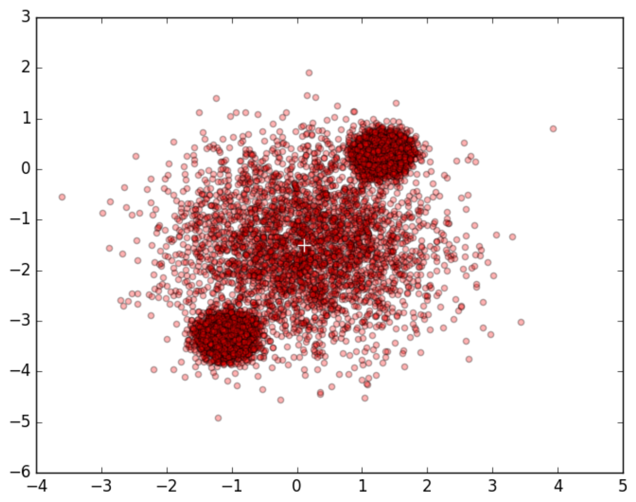
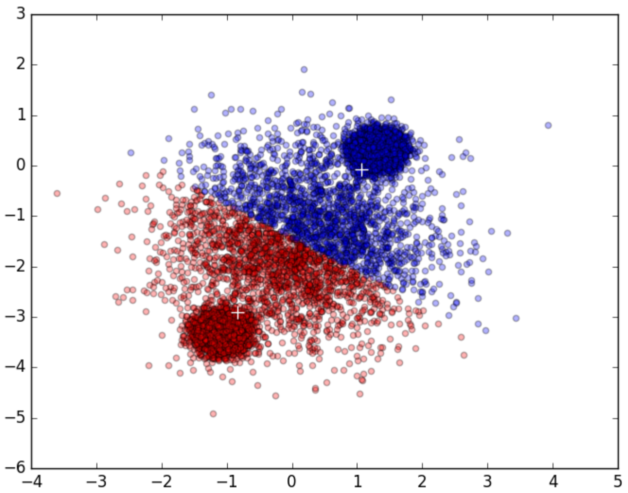
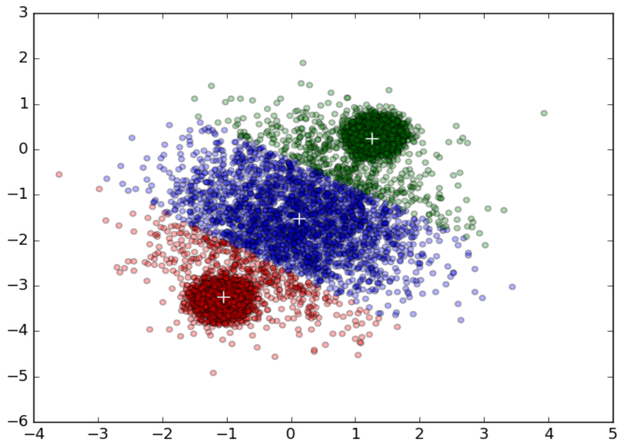
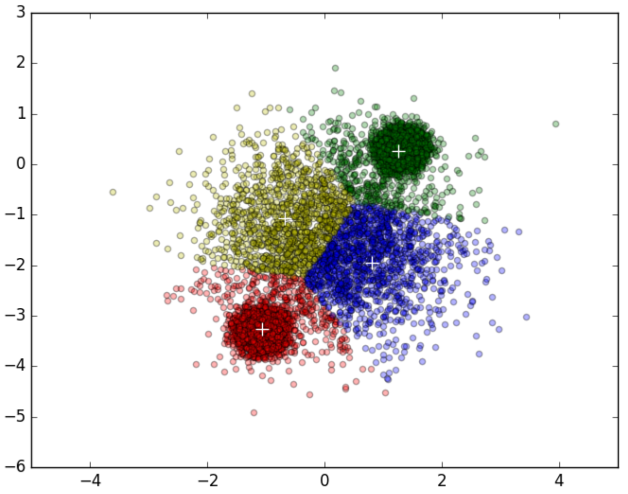


Fig2. The loss Vs. the number of updates for K-mean (K = 3)

We can see from the Fig2. that the loss function will converge to a minimum value after a few hundreds updates.

3. We run the Algorithm with K = 1, 2, 3, 4 and 5. The 2D scatter of the data points and the percentage of the data point belonging to each cluster are listed below:

K = 1 All data points belong to the cluster, the percentage is **100%.**	

K = 2 The percentages of the data points belonging to the two clusters are **50.46%** and **49.54%.**	
K = 3 The percentage of the data points belonging to the three clusters are **38.13%, 23.81%** and **38.06%.**	
K = 4 The percentage of the data points belonging to the four clusters are **37.13%, 13.53%, 37.30%** and **12.04%.**	

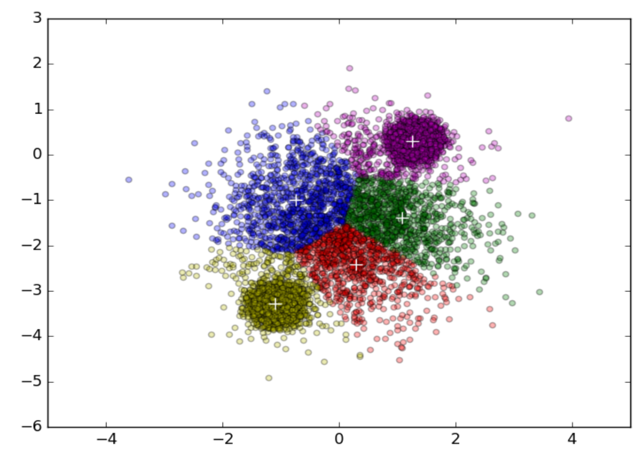
K = 5 The percentage of the data points belonging to the five clusters are **8.74%, 10.73%, 8.64%, 35.77%** and **36.12%.**	

Fig3. 2D scatter plot of data points colored by their cluster assignments

We can see from the above plot that when all data points are divided by two clusters, either side has almost the equal percentage (~50%). It also emphasized the symmetric characteristic of the plot. When K continues increasing, the centroids of two clusters with larger data density remain almost unchanged. While the central part of the data points is divided into the extra clusters according to the distance from data to the centroids. Thus, we think the central part of data points can be labeled as one cluster, and the rest part should be the other two clusters.

Thus, we think if we want to divide the data into clusters with equal number points, k = 2 gives the best result; otherwise, **k = 3** should be the best value for clustering.

4. Our result shows that the loss function for validation data is minimal when k equals to 5. Thus, **K = 5 is the best value if we want to minimize the loss function for validation data**. Actually, the more clusters there are, the less the loss will be. It will be the best scenario if there exist 10000 clusters, so each cluster is corresponding to a data point. In this case, every cluster has the zero loss, and the total one will be zero as well. If K continues growing, the additional cluster will be useless.
   Here are the losses for the validation data when k ranges from 1 to 5:

K	Loss for validation data
1	12969.2
2	3065.8
3	1693.22
4	1100.41
5	934.067

Task 2. Mixtures of Gaussians

2.1 The Gaussian cluster model

1.The expression for the latent variable posterior distribution P(z|x) is:

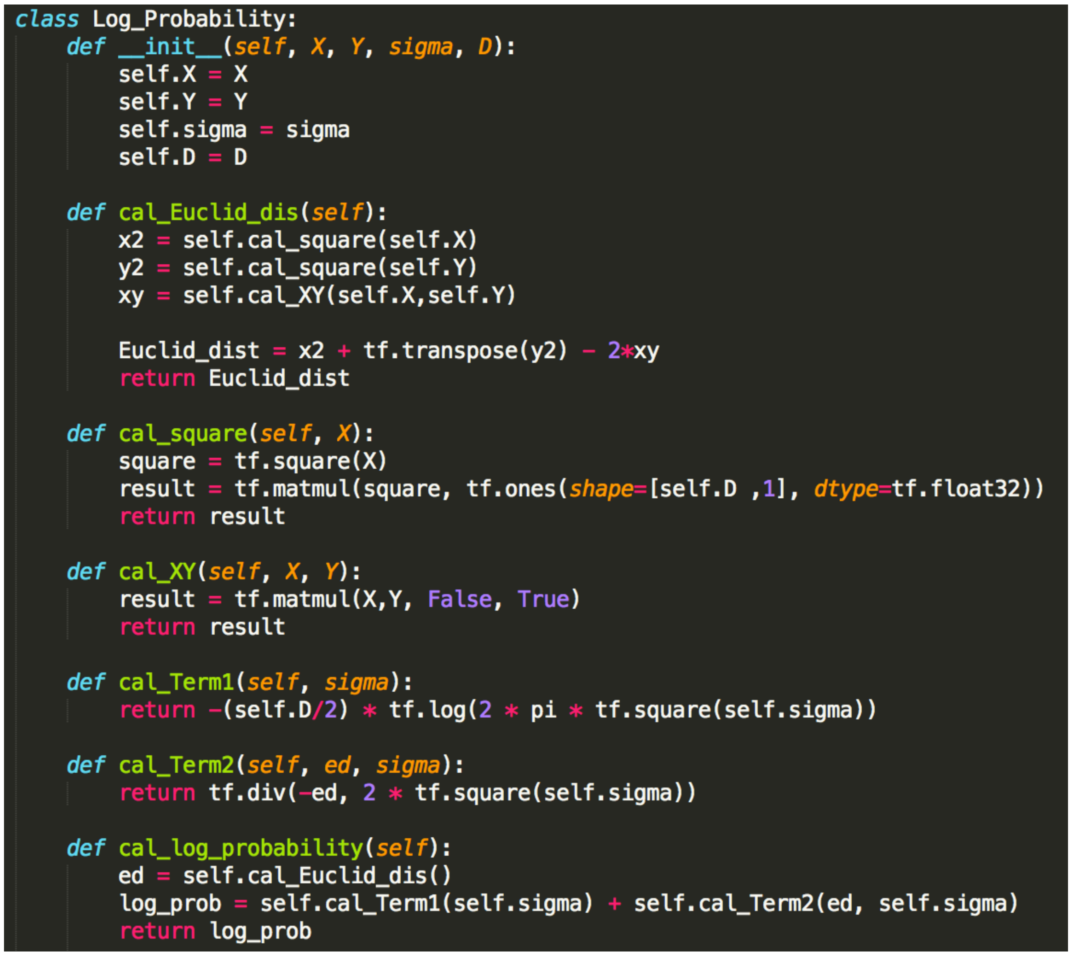
$$P(z=k|x) = \frac{P(x|z=k)P(z=k)}{\Sigma_{z'=1}^{K} P(x|z')P(z')} = \frac{\mathcal{N}(x;\mu^k,{\sigma^k}^2)\pi^k}{\Sigma_{j=1}^{K} \mathcal{N}(x;\mu^j,{\sigma^j}^2)\pi^j} = \frac{\frac{\pi^k}{\sqrt{\left(2\pi{\sigma^k}^2\right)^D}} e^{-\frac{(x-\mu^k)^2}{2{\sigma^k}^2}}}{\Sigma_{j=1}^{K}\left(\frac{\pi^j}{\sqrt{\left(2\pi{\sigma^j}^2\right)^D}} e^{-\frac{(x-\mu^j)^2}{2{\sigma^j}^2}}\right)}$$

where σ is the standard deviation, μ is the mean and D is the dimension of dataset.

2.The log probability density function for cluster k is:

$$log\mathcal{N}(x;\mu^k,{\sigma^k}^2) = log\frac{1}{\sqrt{\left(2\pi{\sigma^k}^2\right)^D}}e^{-(x-\mu^k)^2/2{\sigma^k}^2} = -\frac{D}{2}log(2\pi{\sigma^k}^2) - \frac{(x-\mu^k)^2}{2{\sigma^k}^2}$$

The function is implemented in the "**Log_Probability.py**" file. To get the log probability for all B data points and K clusters, we use the Tensorflow broadcasting. The code is also listed as below:



```python
class Log_Probability:
    def __init__(self, X, Y, sigma, D):
        self.X = X
        self.Y = Y
        self.sigma = sigma
        self.D = D

    def cal_Euclid_dis(self):
        x2 = self.cal_square(self.X)
        y2 = self.cal_square(self.Y)
        xy = self.cal_XY(self.X,self.Y)

        Euclid_dist = x2 + tf.transpose(y2) - 2*xy
        return Euclid_dist

    def cal_square(self, X):
        square = tf.square(X)
        result = tf.matmul(square, tf.ones(shape=[self.D ,1], dtype=tf.float32))
        return result

    def cal_XY(self, X, Y):
        result = tf.matmul(X,Y, False, True)
        return result

    def cal_Term1(self, sigma):
        return -(self.D/2) * tf.log(2 * pi * tf.square(self.sigma))

    def cal_Term2(self, ed, sigma):
        return tf.div(-ed, 2 * tf.square(self.sigma))

    def cal_log_probability(self):
        ed = self.cal_Euclid_dis()
        log_prob = self.cal_Term1(self.sigma) + self.cal_Term2(ed, self.sigma)
        return log_prob
```

Fig4. Implements of log probability density functions for cluster K

3.The log probability of the cluster variable z given the data vector x, P(z|x) is:

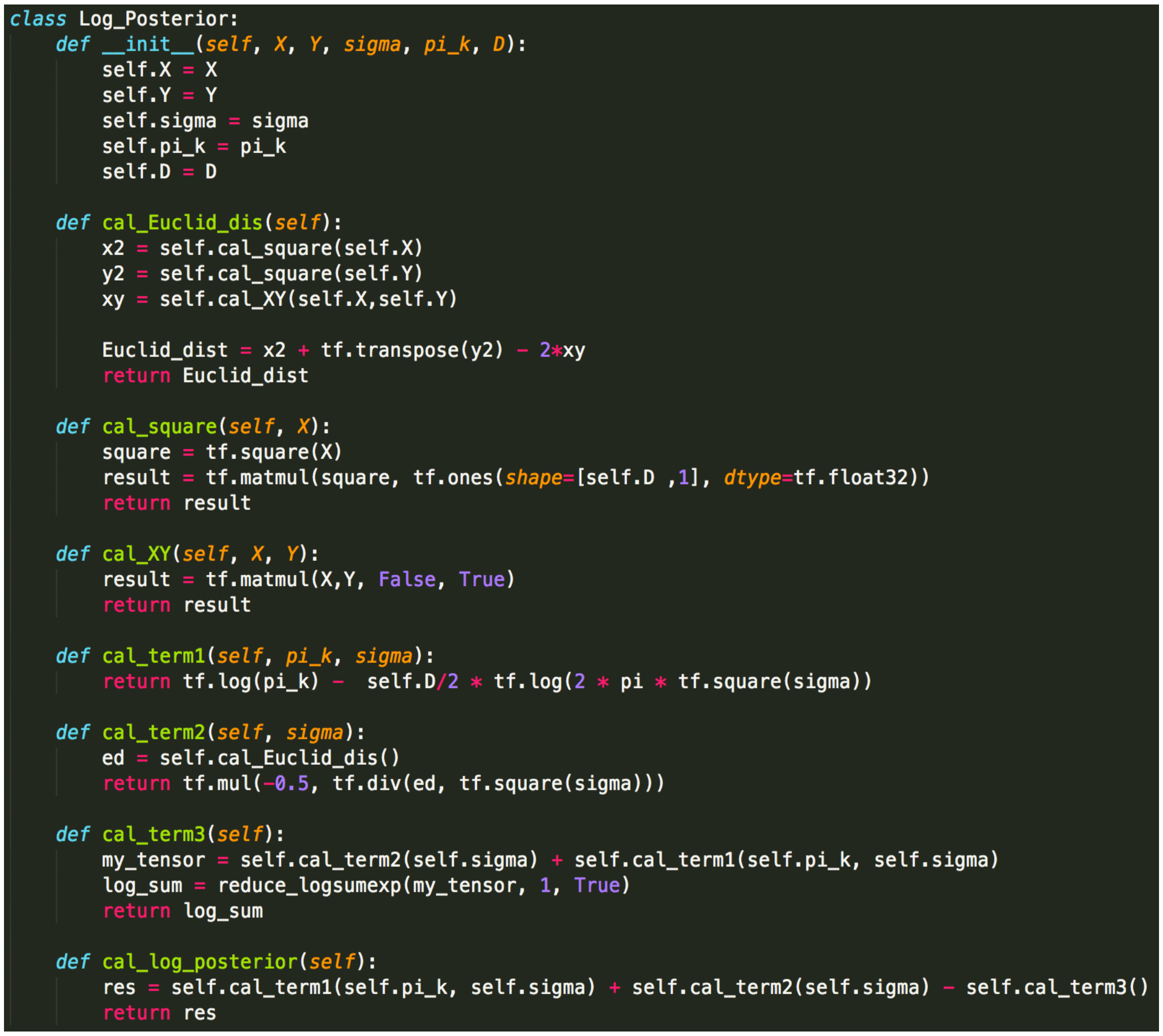
$$logP(z=k|x) = P(x,z=k)/\Sigma_{z'=1}^{K}P(x,z')$$

$$= log\frac{\pi^k\cdot\frac{1}{\sqrt{\left(2\pi\sigma^{k^2}\right)^D}}e^{-\left(x-\mu^k\right)^2/2\sigma^{k^2}}}{\Sigma_{k'=1}^{K}\left(\pi^{k'}\cdot\frac{1}{\sqrt{\left(2\pi\sigma^{k'^2}\right)^D}}e^{-\left(x-\mu^{k'}\right)^2/2\sigma^{k'^2}}\right)}$$

$$= log\frac{e^{\left(-\frac{\left(x-\mu^k\right)^2}{2\sigma^{k^2}}+log\frac{\pi^k}{\sqrt{\left(2\pi\sigma^{k^2}\right)^D}}\right)}}{\Sigma_{k'=1}^{K}e^{\left(-\frac{\left(x-\mu^{k'}\right)^2}{2\sigma^{k'^2}}+log\frac{\pi^{k'}}{\sqrt{\left(2\pi\sigma^{k'^2}\right)^D}}\right)}}$$

$$= -\frac{\left(x-\mu^k\right)^2}{2\sigma^{k^2}} + log\pi^k - \frac{D}{2}log\left(2\pi\sigma^{k^2}\right) - log\Sigma_{k'=1}^{K}e^{\left(-\frac{\left(x-\mu^{k'}\right)^2}{2\sigma^{k'^2}}+log\pi^{k'}-\frac{D}{2}log\left(2\pi\sigma^{k'^2}\right)\right)}$$

where $-\frac{\left(x-\mu^{k'}\right)^2}{2\sigma^{k'^2}} + log\pi^{k'} - \frac{D}{2}log\left(2\pi\sigma^{k'^2}\right)$ is the input tensor of reduce_logsumexp() function. The function is implemented in the "**Log_Posterior.py**" file. The code is also listed below:


```python
class Log_Posterior:
    def __init__(self, X, Y, sigma, pi_k, D):
        self.X = X
        self.Y = Y
        self.sigma = sigma
        self.pi_k = pi_k
        self.D = D

    def cal_Euclid_dis(self):
        x2 = self.cal_square(self.X)
        y2 = self.cal_square(self.Y)
        xy = self.cal_XY(self.X,self.Y)

        Euclid_dist = x2 + tf.transpose(y2) - 2*xy
        return Euclid_dist

    def cal_square(self, X):
        square = tf.square(X)
        result = tf.matmul(square, tf.ones(shape=[self.D ,1], dtype=tf.float32))
        return result

    def cal_XY(self, X, Y):
        result = tf.matmul(X,Y, False, True)
        return result

    def cal_term1(self, pi_k, sigma):
        return tf.log(pi_k) -  self.D/2 * tf.log(2 * pi * tf.square(sigma))

    def cal_term2(self, sigma):
        ed = self.cal_Euclid_dis()
        return tf.mul(-0.5, tf.div(ed, tf.square(sigma)))

    def cal_term3(self):
        my_tensor = self.cal_term2(self.sigma) + self.cal_term1(self.pi_k, self.sigma)
        log_sum = reduce_logsumexp(my_tensor, 1, True)
        return log_sum

    def cal_log_posterior(self):
        res = self.cal_term1(self.pi_k, self.sigma) + self.cal_term2(self.sigma) - self.cal_term3()
        return res
```
Fig5. Implementations of log probability of the cluster variable z given the data vector x

We note that the input tensor may be too small when the standard deviation is very small, such that the exponentiation is 0. When we use the tf.reduce_sum, the return of logarithm could be –inf. Thus, we should use **reduce_logsumexp** here, to ensure that we shift the center of the exponentiated variables, such that the largest value we want to exponentiate is zero. By doing this change, even the rest exponentiations will lead to underflow, we could still get a reasonable value. The whole process is listed as following:

$$\begin{aligned} log\Sigma_{k=1}^{K} e^{x_k} &= log(e^{x_1} + e^{x_2} + \cdots + e^{x_k}) \\ &= log(e^{x_1 - x_m} + e^{x_2 - x_m} + \cdots + e^{0} + \cdots + e^{x_k - x_m})e^{x_m} \\ &= log(e^{x_1 - x_m} + e^{x_2 - x_m} + \cdots + e^{0} + \cdots + e^{x_k - x_m}) + x_m \end{aligned}$$

where the x_m is the maximum value in the input tensor. Even when x_k (for k = 1 to K) is very small, we still can get a reasonable result in this way.

2.2 Learning the MoG

1.

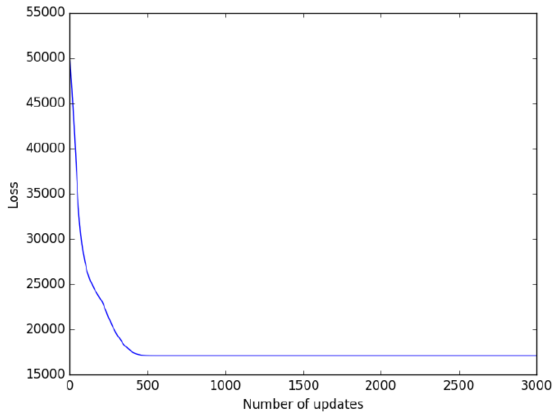
$$\nabla log P(x) = \nabla log \Sigma_{k=1}^{K} P(z=k)P(x|z=k)$$
$$= \frac{\nabla \Sigma_{k=1}^{K} P(z=k)P(x|z=k)}{\Sigma_{k'=1}^{K} P(z=k')P(x|z=k')}$$
$$= \frac{\Sigma_{k=1}^{K} \nabla P(z=k)P(x|z=k)}{\Sigma_{k'=1}^{K} P(z=k')P(x|z=k')}$$
$$= \frac{\Sigma_{k=1}^{K} P(z=k)P(x|z=k)\nabla \log P(z=k)P(x|z=k)}{\Sigma_{k'=1}^{K} P(z=k')P(x|z=k')}$$
$$= \Sigma_{k=1}^{K} \left(\frac{P(z=k)P(x|z=k)}{\Sigma_{k'=1}^{K} P(z=k')P(x|z=k')} \right) \nabla \log P(z=k)P(x|z=k)$$
$$= \Sigma_k P(z=k|x) \nabla \log P(x, z=k)$$

2. We set the learning rate of AdamOptimazer as 0.01 and use the gradient descent to minimize the negative log likelihood. The function is implemented in "**MoG.py**".

When K = 3, we get these model parameters:

μ^k	μ^1[0.10617115, -1.52815366] μ^2[-1.10174823, -3.30640173] μ^3[1.298244, 0.30914176]
π^k	[0.33480254, 0.33164248, 0.33355504]
σ^{k^2}	[0.98666757, 0.03905554, 0.03885973]

The loss Vs. the number of updates are shown in Fig6.



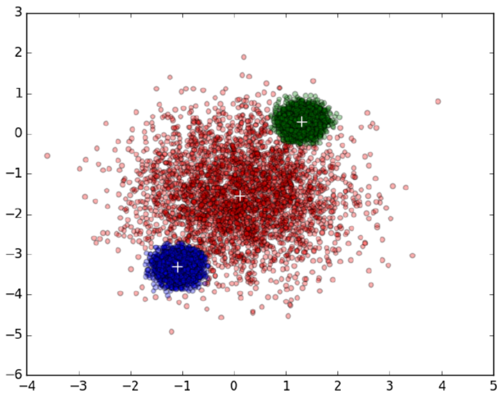
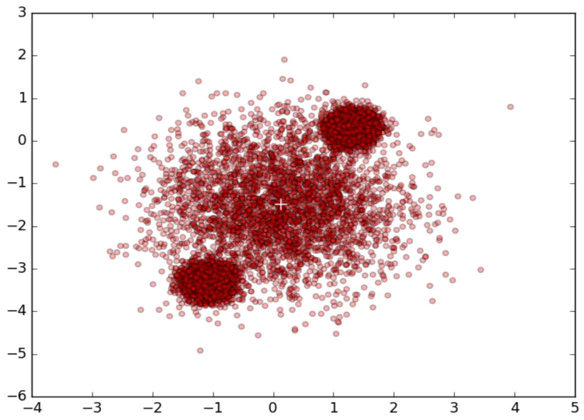
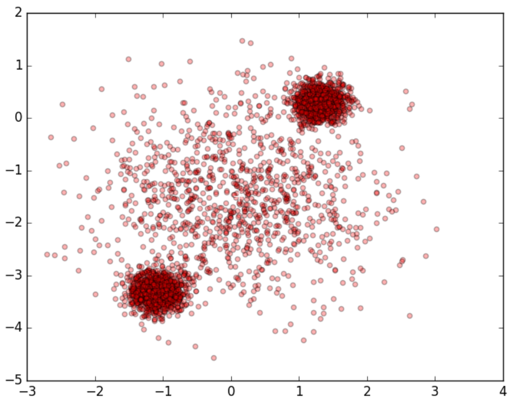
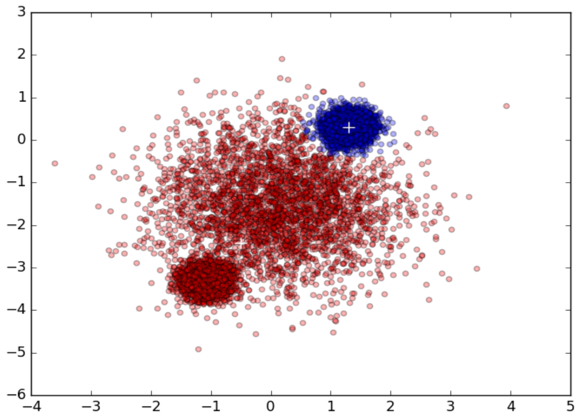
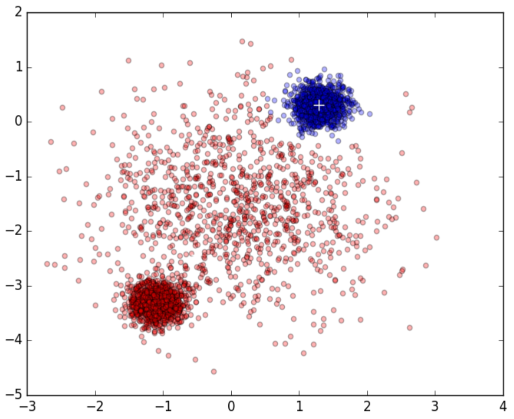
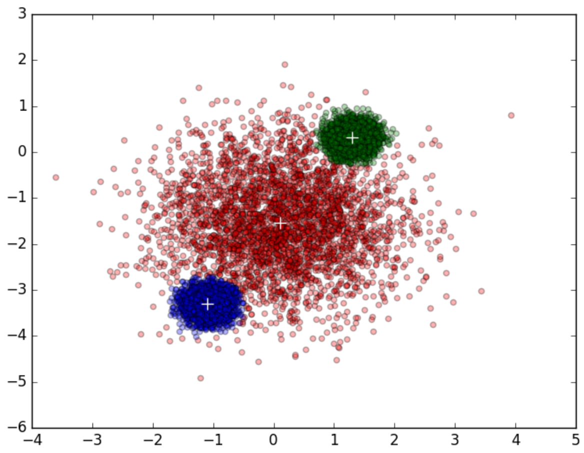
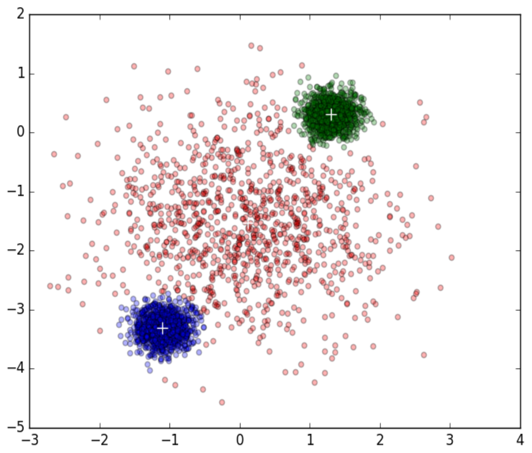


Fig6. The loss Vs. the number of updates for MoG (K = 3) Fig7. 2D scatter of data points (K = 3)

We could see from Fig6 and the model parameters that the loss function converges to 17000 after about 500 times updates. The data points are divided into three clusters. One has a larger standard deviation and the other two clusters have a much smaller standard deviation.

3. When 1/3 of the data is hold for validation and set K from 1 to 5, the MoG mode give us the following result.

	Complete Data	Validation Data
K = 1 Loss for validation data is **11680.0.**		
K = 2 Loss for validation data is **8051.33.**		
K = 3 Loss for validation data is **5584.35**.		

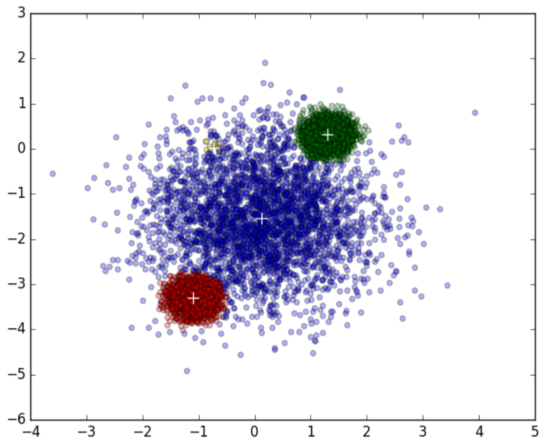
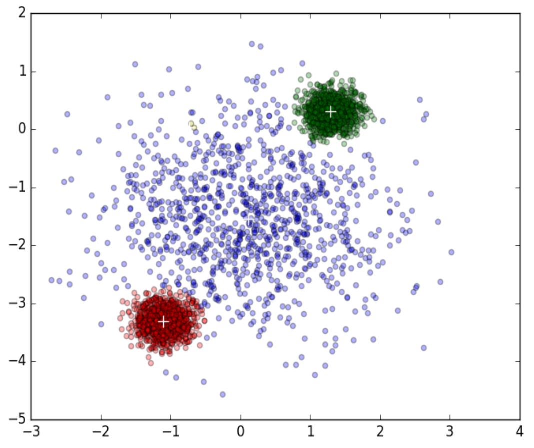
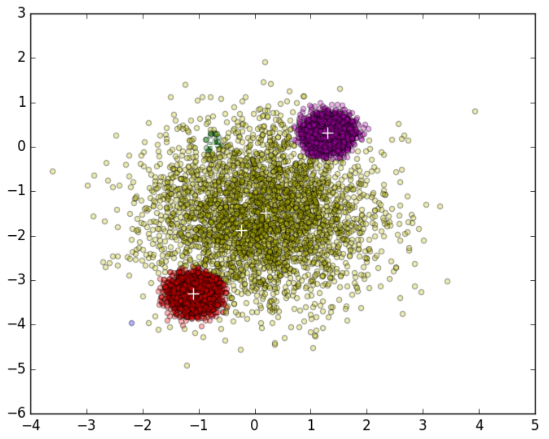
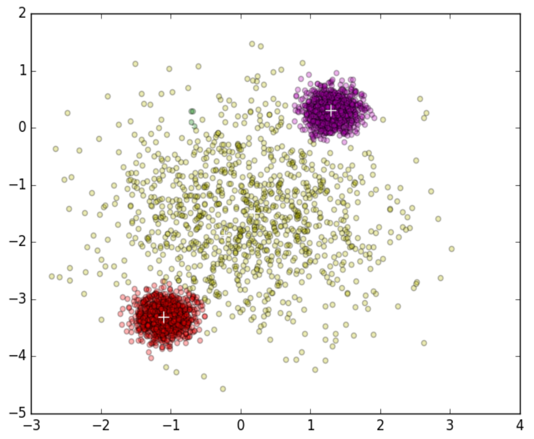
K = 4 Loss for validation data is **5588.47**.		
K = 5 Loss for validation data is **5588.92**.		

Fig8. 2D scatter plot of data points by using MoG for different K values

From the plots and loss function results for validation data, we could conclude that **3** is the best value for clusters to divide the data points. **When K = 3, we receive the minimum loss** and when K continues increasing, the loss will increase a little.

In the scatter plots of data points colored by cluster assignments, we could observe that when K = 3, the data points are evenly divided into three clusters. However, when k increases to 4 and 5, there will be a small number of data points (~10-20) to be assigned to extra clusters. This extra clusters have much smaller π^k, which have little effect on the clustering. Thus, the likelihood is extremely small for data points to be assigned to these extra clusters. By training the MoG model for lots of times, we find that the percentage of data points to be assigned to the fourth and fifth cluster is really small (e.g. ~0.001), sometimes even to be zero. Thus, **we think K = 3 is the best value for the data points by using the MoG method.**

4 When we run both K-means and MoG on the data100D.npy, we got the following result:

Proportion for different clusters in k-means:
(In all these cases, we try many times to get the global minimum)

K	Percentage
1	[1.]
2	[0.20, 0.80]
3	[0.20, 0.40, 0.40]
4	[0.30 , 0.20, 0.30, 0.20]
5	[0.10, 0.20, 0.20, 0.30, 0.20]

Parameters and proportions for different clusters in MoG:
K = 1:

π^k	[1.]
σ^2	[1.00015497]
Loss	1.41893e+06
Cluster Percentage	**[1.]**

K = 2:

π^k	[0.8 0.2]
σ^2	[0.97178531 0.09151918]
Loss	1.17364e+06
Cluster Percentage	**[0.8 0.2]**

K = 3:

π^k	[0.5 0.2 0.3]
σ^2	[0.89462709 0.12682001 0.24339446]
Loss	982988.0
Cluster Percentage	**[0.5 0.2 0.3]**

K = 4:

π^k	[0.2 0.3 0.2 0.3]
σ^2	[0.09153358 0.60416687 0.48652625 0.24351415]
Loss	833978.0
Cluster Percentage	**[0.2 0.3 0.2 0.3]**

K = 5:

π^k	[1.46543883e-07 0.2 0.3 0.2 0.3]
σ^2	[0.28937501 0.09158549 0.2434765 0.12682505 0.78711444]
Loss	739165.0
Cluster Percentage	**[0. 0.2 0.3 0.2 0.3]**

We run the two methods many times, and find that K-means can divide the 100-dimension data into K clusters as long as the loss converges to global minimum. But each cluster may have different percentages of data, which results from the random initialization of centroids.

For the MoG method, the clustering also depends on the initialization of the variables. As the K increases, the loss will decrease accordingly. We find that for k equals to 1, 2, 3, and 4, the 100-dimension data points can be divided into K clusters since the loss function has converged to the global minimum. However, for k equals to 5, the loss can still decrease a little, but there are still 4 clusters in the final results, and the percentage of data points in each cluster is the same with the result when k = 4.

Therefore, we think there are **4 clusters** within the dataset, and the percentage of data points in each cluster is **20%, 30%, 20% and 30%** respectively.